Genetic Algorithms

Sean Peng
Georgia Institute of Technology
speng75@gatech.edu

Ron Guter
Georgia Institute of Technology
rguter3@gatech.edu

Quintin Buckley
Georgia Institute of Technology
qbuckley6@gatech.edu

***Abstract*—This paper explores the effectiveness of Genetic Algorithms (GA) in solving optimization problems, as well as its applications to finance and hybrid GAs that combine GAs with other optimization algorithms. We talk about the mathematical theory behind GAs and implement them from scratch using Python. We first investigate the ability of GAs to solve simple optimization functions. We then explore the potential benefits of hybridizing GAs with Particle Swarm Optimization (PSO) to combine the benefits of each. Finally, we apply a GA to the complex problem of portfolio optimization, using real life data sets.**

***Index Terms*—genetic algorithms, particle swarm optimization, portfolio optimization**

I. Background

Genetic algorithms are a type of optimization algorithm inspired by the theory of evolution through natural selection. Genetic algorithms use simplified representations of biological concepts such as genetics, fitness, reproduction, and mutation, to evolve and optimize potential solutions over several generations [1].

A genetic algorithm's behavior is modified by three parameters which are preset by the user and will be discussed further on. The algorithm begins with the initial population, usually a randomly generated set of candidate solutions. Each candidate solution is represented by a *chromosome*, encoded as an array of values. For instance, if the solution space to the problem has n dimensions, then each candidate solution can be represented as an array: $[p_1, p_2, ..., p_n]$, where p_i is the value of the i^{th} parameter. Most often, parameters, and subsequently chromosomes, are encoded as bit strings.

The *fitness function* is what the genetic algorithm tries to optimize. It dictates the general direction of the evolution process as the chromosomes converge toward an optimal solution. For an algorithm where each chromosome has 2 parameters, a fitness function could be $f(p_1, p_2) = p_1^2 + p_2$, where a higher fitness indicates a better solution.

Some chromosomes in the current generation become parents for the next generation, decided through a *selection operator*. Chromosomes with higher fitness have a higher probability of becoming parents. There are many ways to define the selection operator. For instance, the probability of chromosome C_1 becoming a parent could be:

$$P(C_1) = \frac{f(C_1)}{\sum_{i=1}^{n} f(C_i)},$$

where f is the fitness function. Another approach to the selection operator is *tournament selection*, where each parent is the fittest out of k random chromosomes in the population and k is one of our hyperparameters which takes on a small integer value. A hyperparamter is a parameter external to, and set before, the algorithm.

Parents reproduce by swapping segments of their bit strings, analogous to the recombination of DNA strands in nature. These segments are determined by randomly selected crossover points. For example, let parents $C_1 = 11111, C_2 = 00000$. If we have one crossover point at bit 2, then their child $C_3 = 11000$. The purpose of crossover is to potentially combine the best features of the parents. The rate at which crossover occurs is determined by the hyperparameter, p_c, which takes a decimal value within $[0, 1]$.

Lastly, the *mutation operator* randomly flips bits in a chromosome at a very low probability. The exact rate determined by our last hyperparameter, p_m, which also takes a decimal value within $[0, 1]$. This useful process taken from nature prevents the algorithm from getting stuck at local optima before finding the global optimum. It maintains genetic diversity in the population, but also makes convergence slower. Typically, for a chromosome with n bits, the mutation rate is $\frac{1}{n}$, with only one bit being mutated per chromosome on average.

II. Usefulness of Genetic Algorithms on Difficult Problems

What sets genetic algorithms apart from many other optimization algorithms is that it is a heuristic algorithm, not analytic. Analytic algorithms generally perform some kind of explicit mathematical process in order to find an exact value for an optimal solution. Gradient descent is one such example, which utilizes the gradient (i.e., derivative) of the fitness function to close in on regions where the function is increasing or decreasing (for maximizing or minimizing, respectively). A heuristic algorithm is one that follows looser, more generalized rules to search for solutions, and its rules are more often related to abstract intuition or observed behaviors rather than directly related to any analysis of the fitness function.

Genetic algorithms, as such, are useful for problems that may be hard or impossible to solve via more analytical algorithms (e.g., gradient descent or other differentiation-based algorithms). Such analytical algorithms may struggle when the objective function is not smooth or differentiable or when the objective function is noisy and stochastic (i.e., many

local optima). The random nature of GAs, however, tends to minimize the impact of these problems. While it is possible for a population to concentrate around a local optimum, selection and mutation make it possible to find and concentrate around better solutions within only a few generations. Additionally, where analytical methods can grow very computationally expensive with a large number of parameters, GAs experience relatively little growth in computational cost as they revolve around simple mathematical and bit operations. Many real world optimization problems also have constraints, which add an extra layer of complexity for analytical algorithms.

The nature of genetic algorithms also means, however, that while they tend to find solutions very close to global optima, they are not likely to find the exact optimal solutions for most problems. Close-to-optimal solutions may be satisfactory, but when an exact solution is required, we can simply then follow with an analytical algorithm search in the region around the solution found by the genetic algorithm and avoid converging to non-global optima. This raises the notion that genetic algorithms are very versatile in being combined with and used alongside other optimization algorithms.

Take, for example, particle swarm optimization (PSO), another heuristic algorithm which utilizes a population of various "particles" (solutions) that change and improve iteratively using information gleaned from both individual particles and the whole population. One of PSO's important characteristics is that it allows for balancing the influence of one particle's best result (exploration) against the population's best result (exploitation) [2]. We can use this principle to create a hybrid genetic algorithm which attempts to create more diversity in the population while still maintaining the core behavior of genetic optimization [4]. This can be extremely useful when working with a noisy objective function over a large search space or simply improving performance over a standard genetic algorithm, and is only one way in which genetic algorithms can be easily fused with other techniques and ideas.

The main result of our project will show how genetic algorithms optimize different functions of increasing complexity, how they can be combined with other algorithms or techniques, and how this can be applied to complex problems in finance, specifically portfolio optimization, where standard analytical algorithms may have difficulty. We will start with a simpler multi-variable function to show how solutions improve over generations and verify that the code generates an optimal solution close to the analytical solution. We will also test the genetic algorithm on a more difficult function with multiple noisy local optima. We will then apply a hybrid genetic and particle swarm algorithm (HGAPSO) to the same function and compare its performance to the standard genetic algorithm.

A. Simple Function Optimization

Here is the general procedure our genetic algorithm will follow in all examples:

Algorithm 1 GA

$g = 0$ ▷ Initialize generation 0
P_g = population of n randomly generated chromosomes
Compute $f(p_i)$ for all $p_i \in P_g$ ▷ Compute initial fitness
while population has not converged **do**
  Selection
  Crossover
  Mutation
  $g = g + 1$
  Compute $f(p_i)$ for all $p_i \in P_g$
return fittest chromosome in P_g

Additionally, our implementations of the three basic reproduction operations [1]:

```
def selection(k: int, scores: List[
    float]) -> int:
   selected = np.random.randint(0,
       len(scores), k)
   return selected[np.argmax([scores[
       x] for x in selected])]

def crossover(c1: List[int], c2: List
    [int]) -> Tuple[List[int], List[
    int]]:
   if not c2:
      return c1, c2

   cross_pt = np.random.randint(1,
       len(c1)-1)

   crossed1 = c1[:cross_pt] + c2[
       cross_pt:]
   crossed2 = c2[:cross_pt] + c1[
       cross_pt:]
   return crossed1, crossed2

def mutate(c: List[int]):
   if not c:
      return c
   idx = np.random.randint(len(c))
   c[idx] = 1 - c[idx]
```

This example uses $f(x, y) = 0.01(x - 1)(x + 2)(x + 5)(x - 3)(y - 3)(y + 2)(y + 5)$ as its fitness function (Fig. 1) and attempts to maximize it over the domain $\{(x, y) : x, y \in [-5, 3]\}$. The algorithm ran with 20 chromosomes in the population, tournament selection using $k = 3$, a crossover rate $p_c = 0.8$, and a mutation rate $p_m = 0.1$.

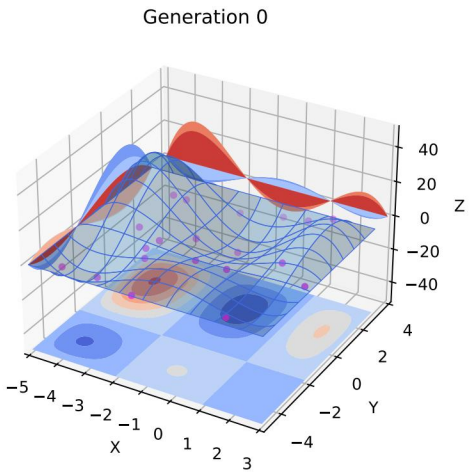

Fig. 1. The initial population.

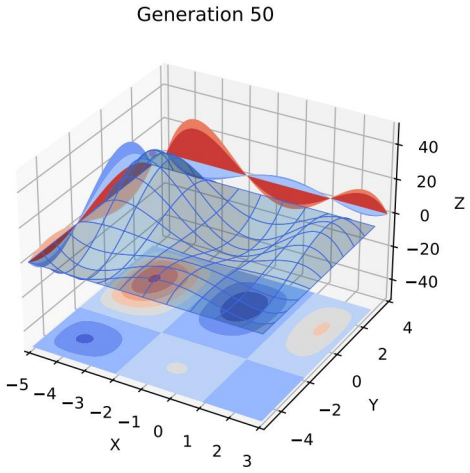

Fig. 2. The final, concentrated population.

Algorithm 2 HGAPSO

$g = 0$ ▷ Initialize generation 0
P_g = population of n randomly generated chromosomes/-particles
V_g = random initial velocity of each particle
Compute $f(p_i)$ for all $p_i \in P_g$ ▷ Compute initial fitness
Record pbest_i for all $p_i \in P_g$ ▷ Fittest point reached by all p_i
gbest = best of all the pbest_i
while population has not converged **do**
  Calculate new V_g values ▷ Particle swarm computation
  Enhance all $p_i \in P_g$ using V_g
  Compute $f(p_i)$ for all $p_i \in P_g$
  Update all the pbest_i
  Update gbest
  C_g = tournament selection for all $p_i \in P_g$ ▷ Select parents
  CV_g = velocities corresponding to each $c_i \in C_g$
  Apply crossover on all $c_i \in C_g$
  Apply arithmetic crossover on all $cv_i \in CV_g$
  Apply mutation on all $c_i \in C_g$
  $P_g = C_g$ ▷ Enhanced and evolved particles
  $V_g = CV_g$
  $g = g + 1$
return fittest chromosome in P_g

Note that since PSO requires the use of velocities for each particle, we also introduce a variation on the previous crossover operator which is more suited for direct application to decimal values:

```
def arithmetic_crossover(c1: float, c2:
    float) -> Tuple[float, float]:
   alpha = np.random.rand()
   return alpha*c1 + (1 - alpha)*c2,
       alpha*c2 + (1-alpha)*c1
```

B. Difficult Function Optimization with HGAPSO

This example uses $f(x, y) = (x - 3.14)^2 + (y - 2.72)^2 + \sin(3x + 1.41) + \sin(4y - 1.73)$ as its fitness function (Fig. 3) and attempts to minimize it over the domain $\{(x, y) : x, y \in [0, 5]\}$. This function is much noisier than the previous example and thus potentially a more difficult problem. We will compare the performance of our standard genetic algorithm with the performance of hybrid algorithm on this problem. The hybrid algorithm follows this general procedure, adapted from the procedure given in [3]:

We also introduce three new hyperparameters, w, c_1, c_2 which influence the enhancement operations. Notably, c_1 and c_2 are known as cognitive and social coefficients, respectively, and correspond to the weight given to exploration versus exploitation, respectively. Both algorithms were run using 20 particles, $k = 3$, $p_c = 0.7$, and $p_m = 0.2$, and HGAPSO was given $w = 0.8$, $c_1 = 0.1$, and $c_2 = 0.1$. The algorithms ran until they either found a solution within one-hundredth distance of the true global minimum or reached 200 iterations.

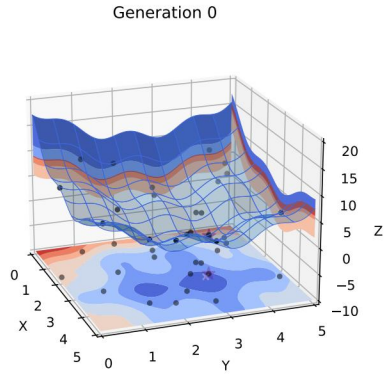

Fig. 3. Standard GA's initial state, also projected onto the contour map.

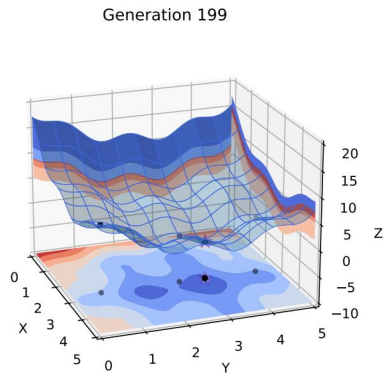

Fig. 4. Standard GA's final state.

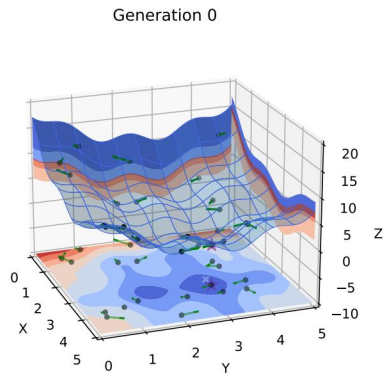

Fig. 5. HGAPSO's initial state, also projected onto the contour map. Arrows represent particle velocity.

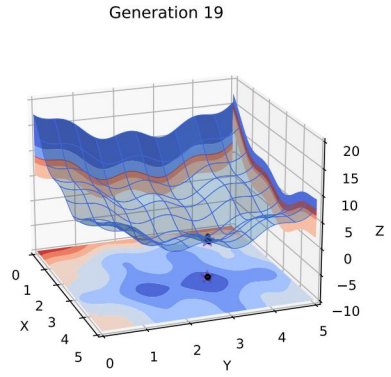

Fig. 6. HGAPSO's final state.

In this instance, the standard GA was able to find a solution *close* to, but not within our termination conditions after ~ 200 generations (Fig. 4). On the other hand, HGAPSO found a satisfactory solution by only generation 19 (Fig. 6). Though this is just one instance, numerous reruns have shown that, on average, HGAPSO is able to generate a satisfactory solution in fewer generations compared to the standard GA.

III. Application of GAs to Portfolio Optimization

A portfolio is a partition of the total capital, where each partition is allocated or a different asset. Let x_i be the fraction of capital allocated for the i^{th} asset. Then, if we consider each parition as a chromosome, then each chromosome must satisfy the constraint that $\sum_{i=1}^{n} x_i = 1$. Therefore, portfolio optimization is classified as a constrained utility maximization problem [5]. Many metrics are important when evaluating a portfolio, including expected return, risk aversion, etc. Therefore, our fitness function needs to take these metrics into account. There are many approaches to portfolio optimization, both analytical and numerical. Some examples include linear and non-linear programming, principal component-based methods, stochastic programming, etc. GAs are good for portfolio optimization because there are often no good analytical solutions. In this section, we will use real life financial data sets and build an optimal portfolio using a genetic algorithm. We will explore stability metrics such as the Sharpe ratio, and deal with constraints in the problem.

We will use data sets from Yahoo Finance to get the stock prices of multiple companies from 2019 to 2021 [6]. We will take data from 12 sources, including AAPL, AMZN, MSFT, TSLA, etc. Here are the first few rows and columns of the compiled Pandas DataFrame.

First, we need to calculate the returns of these assets. The historical return of an asset is defined as the relative difference in stock price from one time period to another. The return of asset i from time t to time t_0 can be calculated as

$$\text{return}_i(t) = \frac{\text{price}_i(t) - \text{price}_i(t_0)}{\text{price}_i(t_0)}$$

TABLE I
SAMPLE OF STOCK DATA SETS FROM YAHOO FINANCE

Date	Stocks					
	ADBE	***AMZN***	***MSFT***	***NVDA***	***TXN***	***IRX***
2019-05-31	270.89	1775.06	121.00	33.73	98.27	2.29
2019-06-01	270.89	1775.06	121.00	33.73	98.27	2.29
2019-06-02	270.89	1775.06	121.00	33.73	98.27	2.29
2019-06-03	259.02	1692.68	117.24	33.31	99.08	2.28
2019-06-04	268.70	1729.56	120.49	35.60	102.81	2.29

For our entire portfolio, with our total capital spread amongst many different assets, the total return can therefore be calculated as

$$\text{total return}(t) = \sum_{i=1}^{n} x_i \cdot \text{price}_i(t)$$

where x_i is the fraction of the total capital that is invested into the i^{th} asset. For this example, we calculated monthly returns and averaged them to use for our genetic algorithm. We will run a genetic algorithm using the expected return of a portfolio as the fitness function. In this case, a chromosome is a list of weights x_i, a specific partition of capital among the assets. Therefore, our initial population are randomly generated lists of weights.

The selection procedure will select the top third of the current generation as parents. The mutation process involves randomly selecting two assets, and transferring a random fraction of capital from one to the other. The crossover procedure for two parent chromosomes a and b is done by selecting a random constant $0 < k < 1$, and computing $k \cdot a + (1-k) \cdot b$, where the factors will be distributed among all the weights in the chromosomes, and addition is performed weight by weight.

We ran the genetic algorithm for 1000 generations, and plotted the average returns for every 10 generations:

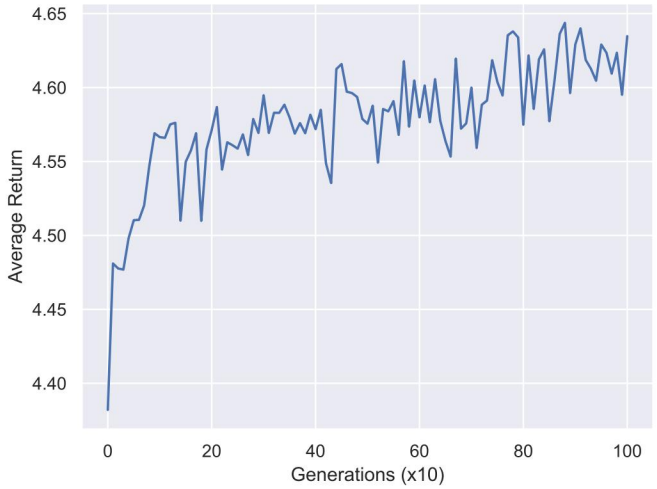

Fig. 7. Average returns for every 10 generations of portfolios

The graph shows that there is a rapid increase in returns up to about the 100th generation, then an overall slower increase in average returns for the later generations. This is to be expected because the first few generations are largely still randomly generated. We can extract the weights of the best performing chromosome according to this fitness function:

TABLE II
BEST CHROMOSOME FOR MAX RETURN

Stock	Weight for Max Return
AAPL	0.12
ADBE	0.01
AMZN	0.06
BTC-USD	0.07
FB	0.12
GC=F	0.05
IRX	0.17
MSFT	0.21
NVDA	0.06
QCOM	0.03
TSLA	0.03
TXN	0.08

This chromosome was part of generation 894, and tells us how we should partition our total capital among these 12 stocks to maximize our expected return. However, return is not the only metric important in optimizing a portfolio. There is always risk when aiming for high returns, called the risk-return tradeoff. One measure of risk is volatility, which is the standard deviation over time, calculated as: $vol = \sigma\sqrt{T}$, where σ is the standard deviation and T is the time window. Just as how we calculated returns, we will calculate volatilities by month, then average them. Using the same generations we has before, for every 10 generations, we plot the average return against the average volatility:

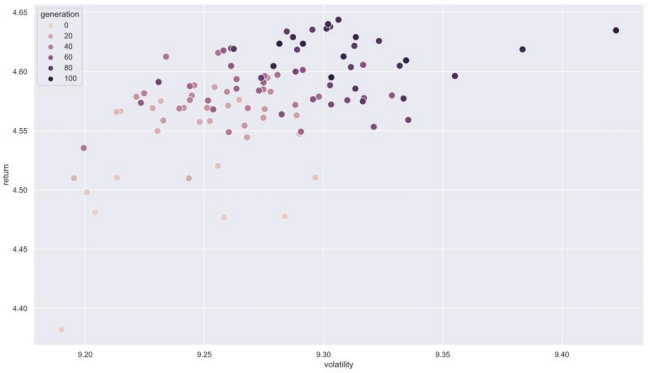

Fig. 8. Risk-return tradeoff

The higher the generation number, the higher the average return. However, the volatility and risk also increases. Therefore, we need a better metric for the fitness function. One commonly used metric is the Sharpe Ratio, proposed by economist William F. Sharpe, who called it the reward-to-variability-ratio:

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

where R_p is the expected return of the portfolio, σ_p is the standard deviation of the returns, and R_f is the risk-free rate. The risk-free rate is the theoretical rate of return for an investment with zero risk. The yield on a three-month T-bill is usually used as the risk-free rate. We will use 1.2 as an approximate over our time range. By running our genetic algorithm on this new fitness function, we were able to achieve 0.5738 Sharpe ratio, with a 6.21% return. A Sharpe ratio over 0.5 is a sign of a well diversified portfolio of stocks and bonds.

These results good but not exceptional, this is mainly due to our consideration of only 12 different stocks. Additionally, there are several advanced financial strategies that can be used alongside GAs, such as risk parity strategies, that would improve resutls. The specific weights can be extracted in the same way as in Table II, and we can make a corresponding investment into these stocks.

IV. Conclusion

In summary, genetic algorithms are a powerful way of solving optimization problems by evolving a candidate solution set towards the global optimum. We illustrate the strength of genetic algorithms by visualizing the evolution process and showing how candidate solutions improve their fitness over generations. We then apply these techniques to portfolio optimization, using online financial data sets and creating a genetic algorithm that produces an optimal partition of capital according to some metric, such as the Sharpe ratio.

As we have seen, genetic algorithms are a powerful choice compared to other optimization methods like gradient descent, when the objective function to maximize or minimize is not easily differentiable, or has many parameters. They are even able to solve problems with multiple objective functions or constraints. Additionally, for functions with multiple local minima, methods like gradient descent get stuck at these points, whereas the randomness of GAs allows them to find the global minimum as long as there are enough generations. Because of GAs' versatility in solving complex problems with multiple constraints and parameters, they are used in a variety of fields including engineering, medicine, and finance.

V. Appendix

A. GA Implementation

```
def iterate(self):
    # Evaluate this generation
    decoded_pop = [self.decode(chrom) for
         chrom in self.population]
    self.scores = [self.fitness(
        decoded_chrom) for decoded_chrom
        in decoded_pop]
    self.best = (self.decode(self.
        population[np.argmax(self.scores)
        ]), np.max(self.scores))

    # Begin generating the next
        generation via selection,
        crossover, and mutation
    parents = [self.population[selection(
        self.k, self.scores)] for _ in
        range(self.pop_size)]
    new_gen = []
    for x in range(0, len(parents), 2):
        c1 = parents[x]
        c2 = None if x+1 >= len(parents)
            else parents[x+1]

        if np.random.random() < self.p_c:
            c1, c2 = crossover(c1, c2)

        if np.random.random() < self.p_m:
            mutate(c1)
        if np.random.random() < self.p_m:
            mutate(c2)

        new_gen.append(c1)
        # This will only not trigger when
            pop_size is odd and we are on
            the last set of parents
        if c2:
            new_gen.append(c2)

    self.population = new_gen
    self.generation = self.generation + 1
```

B. HGAPSO Implementation Snippets

```
def iterate_hgapso(self):
   # Enhance all the particles
   particles_dec = np.asarray([decode(
       particle) for particle in self.
       particles])
   pbest_dec = np.asarray([decode(pbest)
        for pbest in self.pbest])
   gbest_dec = decode(self.gbest)
   # Update velocities via the formula:
   # Vi(t+1) = w*Vi(t) + c1r1*(pbesti -
       Xi(t)) + c2r2(gbest - Xi(t))
   ...

   # Update scores, pbests, and gbest
   self.particles = np.asarray([encode(
       particle) for particle in
       particles_dec])
   self.scores = np.asarray([f(particle)
        for particle in particles_dec])
   for i in range(self.pop_size):
      if self.scores[i] < self.
          pbest_scores[i]:
         self.pbest_scores[i] = self.
             scores[i]
         self.pbest[i] = self.particles[
             i]

   ...

   # Now perform tournament selection to
        select parents
   selected_inds = np.asarray([selection
       (self.k, self.scores) for _ in
       range(self.pop_size)])
   parents = self.particles.copy()[
       selected_inds]
   parent_vels = self.vels.copy()[
       selected_inds]
   offspring = np.empty((self.pop_size,
       32), dtype=np.int8)
   offspring_vels = np.zeros((self.
       pop_size, 2))
   for i in range(0, len(parents), 2):
      cross1 = parents[i]
      cross2 = None if i+1 >= len(
          parents) else parents[i+1]
      ...

   # Update scores again
   self.scores = np.asarray([f(particle)
        for particle in particles_dec])
   for i in range(self.pop_size):
      if self.scores[i] < self.
          pbest_scores[i]:
         self.pbest_scores[i] = self.
             scores[i]
         self.pbest[i] = self.particles[
             i]
   ...

   # Increment generation
   self.gen += 1
```

REFERENCES

[1] V. Mallawaarachchi, "Introduction to Genetic Algorithms — Including Example Code," Medium, Mar. 01, 2020. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[2] A. Tam, "A Gentle Introduction to Particle Swarm Optimization," MachineLearningMastery.com, Oct. 11, 2021. https://machinelearningmastery.com/a-gentle-introduction-to-particle-swarm-optimization/

[3] A. Ali and M. A. Tawhid, "A hybrid particle swarm optimization and genetic algorithm with population partitioning for large scale optimization problems," Ain Shams Engineering Journal, vol. 8, no. 2, pp. 191–206, Jun. 2017, doi: 10.1016/j.asej.2016.07.008.

[4] "A hybrid of genetic algorithm and particle swarm optimization for recurrent network design," IEEE Journals & Magazine — IEEE Xplore, Apr. 01, 2004. https://ieeexplore.ieee.org/document/1275532

[5] "Portfolio Optimization," MATLAB & Simulink. https://www.mathworks.com/discovery/portfolio-optimization.html

[6] "Yahoo Finance Data Sets," Yahoo Finance. https://help.yahoo.com/kb/SLN2311.html